Chapter 5

Class Design

OCA EXAM OBJECTIVES COVERED IN THIS CHAPTER:

✓ **Working with Inheritance**

- Describe inheritance and its benefits
- Develop code that demonstrates the use of polymorphism; including overriding and object type versus reference type
- Determine when casting is necessary
- Use super and this to access objects and constructors
- Use abstract classes and interfaces

In Chapter 1, "Java Building Blocks," we introduced the basic definition for a class in Java. In Chapter 4, "Methods and Encapsulation," we delved into constructors, methods, and modifiers, and showed how you can use them to build more structured classes. In this chapter, we'll take things one step further and show how class structure is one of the most powerful features in the Java language.

At its core, proper Java class design is about code reusability, increased functionality, and standardization. For example, by creating a new class that extends an existing class, you may gain access to a slew of inherited primitives, objects, and methods. Alternatively, by designing a standard interface for your application, you ensure that any class that implements the interface has certain required methods defined. Finally, by creating abstract class definitions, you're defining a platform that other developers can extend and build on top of.

Introducing Class Inheritance

When creating a new class in Java, you can define the class to inherit from an existing class. *Inheritance* is the process by which the new child subclass automatically includes any `public` or `protected` primitives, objects, or methods defined in the parent class.

For illustrative purposes, we refer to any class that inherits from another class as a *child class*, or a descendent of that class. Alternatively, we refer to the class that the child inherits from as the *parent class*, or an ancestor of the class. If child X inherits from class Y, which in turn inherits from class Z, then X would be considered an indirect child, or descendent, of class Z.

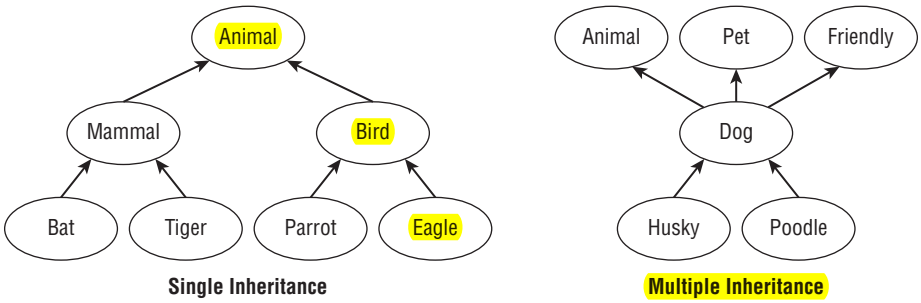
Java supports *single inheritance*, by which a class may inherit from only one direct parent class. Java also supports multiple levels of inheritance, by which one class may extend another class, which in turn extends another class. You can extend a class any number of times, allowing each descendent to gain access to its ancestor's members.

To truly understand single inheritance, it may helpful to contrast it with *multiple inheritance*, by which a class may have multiple direct parents. By design, Java doesn't support multiple inheritance in the language because studies have shown that multiple inheritance can lead to complex, often difficult-to-maintain code. Java does allow one exception to the single inheritance rule: classes may implement multiple interfaces, as you'll see later in this chapter.

Figure 5.1 illustrates the various types of inheritance models. The items on the left are considered single inheritance because each child has exactly one parent. You may notice that single inheritance doesn't preclude parents from having multiple children. The right

side shows items that have multiple inheritance. For example, a dog object has multiple parent designations. Part of what makes multiple inheritance complicated is determining which parent to inherit values from in case of a conflict. For example, if you have an object or method defined in all of the parents, which one does the child inherit? There is no natural ordering for parents in this example, which is why Java avoids these issues by disallowing multiple inheritance altogether.

FIGURE 5.1 Types of inheritance

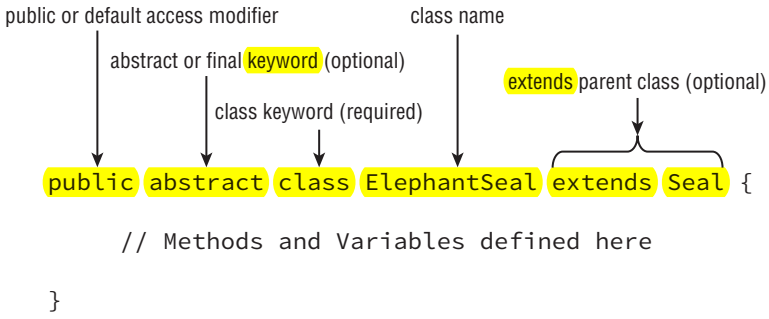


It is possible in Java to prevent a class from being extended by marking the class with the `final` modifier. If you try to define a class that inherits from a `final` class, the compiler will throw an error and not compile. Unless otherwise specified, throughout this chapter you can assume the classes we work with are not marked as `final`.

Extending a Class

In Java, you can extend a class by adding the parent class name in the definition using the `extends` keyword. The syntax of defining and extending a class is shown in Figure 5.2.

FIGURE 5.2 Defining and extending a class



We'll discuss what it means for a class to be `abstract` and `final`, as well as the class access modifiers, later in this chapter.

Because Java allows only one `public` class per file, we can create two files, `Animal.java` and `Lion.java`, in which the `Lion` class extends the `Animal` class. Assuming they are in the same package, an `import` statement is not required in `Lion.java` to access the `Animal` class.

Here are the contents of `Animal.java`:

```
public class Animal {
  private int age;
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
}
```

And here are the contents of `Lion.java`:

```
public class Lion extends Animal {
  private void roar() {
    System.out.println("The "+getAge()+" year old lion says: Roar!");
  }
}
```

Notice the use of the `extends` keyword in `Lion.java` to indicate that the `Lion` class extends from the `Animal` class. In this example, we see that `getAge()` and `setAge()` are accessible by subclass `Lion`, because they are marked as `public` in the parent class. The primitive `age` is marked as `private` and therefore not accessible from the subclass `Lion`, as the following would not compile:

```
public class Lion extends Animal {
  private void roar() {
    System.out.println("The "+age+" year old lion says: Roar!");
    // DOES NOT COMPILE
  }
}
```

Despite the fact that `age` is inaccessible by the child class, if we have an instance of a `Lion` object, there is still an `age` value that exists within the instance. The `age` value just cannot be directly referenced by the child class nor any instance of the class. In this manner, the `Lion` object is actually "bigger" than the `Animal` object in the sense that it includes all the properties of the `Animal` object (although not all of those properties may be directly accessible) along with its own set of `Lion` attributes.

Applying Class Access Modifiers

As discussed in Chapter 4, you can apply access modifiers (`public`, `private`, `protected`, `default`) to both class methods and variables. It probably comes as no surprise that you can also apply access modifiers to class definitions, since we have been adding the `public` access modifier to nearly every class up to now.

For the OCA exam, you should only be familiar with `public` and default package-level class access modifiers, because these are the only ones that can be applied to top-level classes within a Java file. The `protected` and `private` modifiers can only be applied to inner classes, which are classes that are defined within other classes, but this is well out of scope for the OCA exam.

The `public` access modifier applied to a class indicates that it can be referenced and used in any class. The default package private modifier, which is the lack of any access modifier, indicates the class can be accessed only by a subclass or class within the same package.

As you know, a Java file can have many classes but at most one `public` class. In fact, it may have no `public` class at all. One feature of using the default package private modifier is that you can define many classes within the same Java file. For example, the following definition could appear in a single Java file named `Groundhog.java`, since it contains only one `public` class:

```
class Rodent {}

public class Groundhog extends Rodent {}
```

If we were to update the `Rodent` class with the `public` access modifier, the `Groundhog.java` file would not compile unless the `Rodent` class was moved to its own `Rodent.java` file.

The rules for applying class access modifiers are identical for interfaces. There can be at most one `public` class or interface in a Java file. Like classes, top-level interfaces can also be declared with the `public` or default modifiers. We'll discuss interfaces in detail later in this chapter.

For simplicity, any time you see multiple `public` classes or interfaces defined in the same code block in this chapter, assume each class is defined in its own Java file.

Creating Java Objects

Throughout our discussion of Java in this book, we have thrown around the word *object* numerous times—and with good reason. In Java, all classes inherit from a single class,

`java.lang.Object`. Furthermore, `java.lang.Object` is the only class that doesn't have any parent classes.

You might be wondering, "None of the classes I've written so far extend `java.lang.Object`, so how do all classes inherit from it?" The answer is that the compiler has been automatically inserting code into any class you write that doesn't extend a specific class. For example, consider the following two equivalent class definitions:

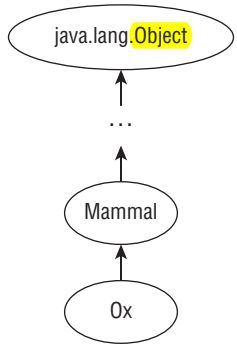
```
public class Zoo {
}

public class Zoo extends java.lang.Object {
}
```

The key is that when Java sees you define a class that doesn't extend another class, it immediately adds the syntax `extends java.lang.Object` to the class definition.

If you define a new class that extends an existing class, Java doesn't add this syntax, although the new class still inherits from `java.lang.Object`. Since all classes inherit from `java.lang.Object`, extending an existing class means the child automatically inherits from `java.lang.Object` by construction. This means that if you look at the inheritance structure of any class, it will always end with `java.lang.Object` on the top of the tree, as shown in Figure 5.3.

FIGURE 5.3 Java object inheritance



All objects inherit java.lang.Object

Defining Constructors

As you may recall from Chapter 4, every class has at least one constructor. In the case that no constructor is declared, the compiler will automatically insert a default no-argument constructor. In the case of extending a class, though, things are a bit more interesting.

In Java, the first statement of every constructor is either a call to another constructor within the class, using `this()`, or a call to a constructor in the direct parent class, using

super(). If a parent constructor takes arguments, the super constructor would also take arguments. For simplicity in this section, we refer to the super() command as any parent constructor, even those that take an argument. Notice the user of both super() and super(age) in the following example:

```
public class Animal {
  private int age;
  public Animal(int age) {
    super();
    this.age = age;
  }
}

public class Zebra extends Animal {
  public Zebra(int age) {
    super(age);
  }
  public Zebra() {
    this(4);
  }
}
```

In the first class, Animal, the first statement of the constructor is a call to the parent constructor defined in java.lang.Object, which takes no arguments. In the second class, Zebra, the first statement of the first constructor is a call to Animal's constructor, which takes a single argument. The class Zebra also includes a second no-argument constructor that doesn't call super() but instead calls the other constructor within the Zebra class using this(4).

Like the this() command that you saw in Chapter 4, the super() command may only be used as the first statement of the constructor. For example, the following two class definitions will not compile:

```
public class Zoo {
  public Zoo() {
    System.out.println("Zoo created");
    super(); // DOES NOT COMPILE
  }
}

public class Zoo {
  public Zoo() {
    super();
    System.out.println("Zoo created");
```

```
    super();  // DOES NOT COMPILE
  }
}
```

The first class will not compile because the call to the parent constructor must be the first statement of the constructor, not the second statement. In the second code snippet, `super()` is the first statement of the constructor, but it also used as the third statement. Since `super()` can only be used as the first statement of the constructor, the code will likewise not compile.

If the parent class has more than one constructor, the child class may use any valid parent constructor in its definition, as shown in the following example:

```
public class Animal {
  private int age;
  private String name;
  public Animal(int age, String name) {
    super();
    this.age = age;
    this.name = name;
  }
  public Animal(int age) {
    super();
    this.age = age;
    this.name = null;
  }
}

public class Gorilla extends Animal {
  public Gorilla(int age) {
    super(age,"Gorilla");
  }
  public Gorilla() {
    super(5);
  }
}
```

In this example, the first child constructor takes one argument, `age`, and calls the parent constructor, which takes two arguments, `age` and `name`. The second child constructor takes no arguments, and it calls the parent constructor, which takes one argument, `age`. In this example, notice that the child constructors are not required to call matching parent constructors. Any valid parent constructor is acceptable as long as the appropriate input parameters to the parent constructor are provided.

Understanding Compiler Enhancements

Up to now, we defined numerous classes that did not explicitly call the parent constructor via the `super()` keyword, so why did the code compile? The answer is that the Java compiler automatically inserts a call to the no-argument constructor `super()` if the first statement is not a call to the parent constructor. For example, the following three class and constructor definitions are equivalent, because the compiler will automatically convert them all to the last example:

```
public class Donkey {
}

public class Donkey {
  public Donkey() {
  }
}

public class Donkey {
  public Donkey() {
    super();
    }
}
```

Make sure you understand the differences between these three `Donkey` class definitions and why Java will automatically convert them all to the last definition. Keep the process the Java compile performs in mind as we discuss the next few examples.

What happens if the parent class doesn't have a no-argument constructor? Recall that the no-argument constructor is not required and only inserted if there is no constructor defined in the class. In this case, the Java compiler will not help and you must create at least one constructor in your child class that explicitly calls a parent constructor via the `super()` command. For example, the following code will not compile:

```
public class Mammal {
  public Mammal(int age) {
  }
}

public class Elephant extends Mammal {  // DOES NOT COMPILE
}
```

In this example no constructor is defined within the `Elephant` class, so the compiler tries to insert a default no-argument constructor with a `super()` call, as it did in the `Donkey` example. The compiler stops, though, when it realizes there is no parent constructor that takes no arguments. In this example, we must explicitly define at least one constructor, as in the following code:

```
public class Mammal {
  public Mammal(int age) {
  }
}

public class Elephant extends Mammal {
  public Elephant() {  // DOES NOT COMPILE
  }
}
```

This code still doesn't compile, though, because the compiler tries to insert the no-argument `super()` as the first statement of the constructor in the `Elephant` class, and there is no such constructor in the parent class. We can fix this, though, by adding a call to a parent constructor that takes a fixed argument:

```
public class Mammal {
  public Mammal(int age) {
  }
}

public class Elephant extends Mammal {
  public Elephant() {
    super(10);
  }
}
```

This code will compile because we have added a constructor with an explicit call to a parent constructor. Note that the class `Elephant` now has a no-argument constructor even though its parent class `Mammal` doesn't. Subclasses may define no-argument constructors even if their parent classes do not, provided the constructor of the child maps to a parent constructor via an explicit call of the `super()` command.

You should be wary of any exam question in which the parent class defines a constructor that takes arguments and doesn't define a no-argument constructor. Be sure to check that the code compiles before answering a question about it.

Reviewing Constructor Rules

Let's review the rules we covered in this section.

Constructor Definition Rules:

1. The first statement of every constructor is a call to another constructor within the class using `this()`, or a call to a constructor in the direct parent class using `super()`.
2. The `super()` call may not be used after the first statement of the constructor.

3. If no `super()` call is declared in a constructor, Java will insert a no-argument `super()` as the first statement of the constructor.
4. If the parent doesn't have a no-argument constructor and the child doesn't define any constructors, the compiler will throw an error and try to insert a default no-argument constructor into the child class.
5. If the parent doesn't have a no-argument constructor, the compiler requires an explicit call to a parent constructor in each child constructor.

Make sure you understand these rules; the exam will often provide code that breaks one or many of these rules and therefore doesn't compile.

Calling Constructors

Now that we have covered how to define a valid constructor, we'll show you how Java calls the constructors. In Java, the parent constructor is always executed before the child constructor. For example, try to determine what the following code outputs:

```
class Primate {
  public Primate() {
    System.out.println("Primate");
  }
}

class Ape extends Primate {
  public Ape() {
    System.out.println("Ape");
  }
}

public class Chimpanzee extends Ape {
  public static void main(String[] args) {
    new Chimpanzee();
  }
}
```

The compiler first inserts the `super()` command as the first statement of both the `Primate` and `Ape` constructors. Next, the compiler inserts a default no-argument constructor in the `Chimpanzee` class with `super()` as the first statement of the constructor. The code will execute with the parent constructors called first and yields the following output:

```
Primate
Ape
```

The exam creators are fond of questions similar to the previous one that try to get you to determine the output of statements involving constructors. Just remember to "think like the compiler" as much as possible and insert the missing constructors or statements as needed.

Calling Inherited Class Members

Java classes may use any `public` or `protected` member of the parent class, including methods, primitives, or object references. If the parent class and child class are part of the same package, the child class may also use any default members defined in the parent class. Finally, a child class may never access a `private` member of the parent class, at least not through any direct reference. As you saw in the first example in this chapter, a `private` member age may be accessed indirectly via a `public` or `protected` method.

To reference a member in a parent class, you can just call it directly, as in the following example with the output function `displaySharkDetails()`:

```
class Fish {
  protected int size;
  private int age;

  public Fish(int age) {
    this.age = age;
  }

  public int getAge() {
    return age;
  }
}

public class Shark extends Fish {

  private int numberOfFins = 8;

  public Shark(int age) {
    super(age);
    this.size = 4;
  }

  public void displaySharkDetails() {
    System.out.print("Shark with age: "+getAge());
```

```
    System.out.print(" and "+size+" meters long");
    System.out.print(" with "+numberOfFins+" fins");
  }
}
```

In the child class, we use the `public` method `getAge()` and `protected` member `size` to access values in the parent class.

As you may remember from Chapter 4, you can use the keyword `this` to access a member of the class. You may also use `this` to access members of the parent class that are accessible from the child class, since a child class inherits all of its parent members. Consider the following alternative definition to the `displaySharkDetails()` method in the previous example:

```
public void displaySharkDetails() {
  System.out.print("Shark with age: "+this.getAge());
  System.out.print(" and "+this.size+" meters long");
  System.out.print(" with "+this.numberOfFins+" fins");
}
```

In Java, you can explicitly reference a member of the parent class by using the `super` keyword, as in the following alternative definition of `displaySharkDetails()`:

```
public void displaySharkDetails() {
  System.out.print("Shark with age: "+super.getAge());
  System.out.print(" and "+super.size+" meters long");
  System.out.print(" with "+this.numberOfFins+" fins");
}
```

In the previous example, we could use `this` or `super` to access a member of the parent class, but is the same true for a member of the child class? Consider this example:

```
public void displaySharkDetails() {
  System.out.print("Shark with age: "+super.getAge());
  System.out.print(" and "+super.size+" meters long");
  System.out.print(" with "+super.numberOfFins+" fins"); // DOES NOT COMPILE
}
```

This code will not compile because `numberOfFins` is only a member of the current class, not the parent class. In other words, we see that `this` and `super` may both be used for methods or variables defined in the parent class, but only `this` may be used for members defined in the current class.

As you'll see in the next section, if the child class overrides a member of the parent class, `this` and `super` could have very different effects when applied to a class member.

super() vs. *super*

As discussed in Chapter 4, `this()` and `this` are unrelated in Java. Likewise, `super()` and `super` are quite different but may be used in the same methods on the exam. The first, `super()`, is a statement that explicitly calls a parent constructor and may only be used in the first line of a constructor of a child class. The second, `super`, is a keyword used to reference a member defined in a parent class and may be used throughout the child class.

The exam may try to trick you by using both `super()` and `super` in a constructor. For example, consider the following code:

```
public Rabbit(int age) {
  super();
  super.setAge(10);
}
```

The first statement of the constructor calls the parent's constructor, whereas the second statement calls a function defined in the parent class. Contrast this with the following code, which doesn't compile:

```
public Rabbit(int age) {
  super;  // DOES NOT COMPILE
  super().setAge(10);  // DOES NOT COMPILE
}
```

This code looks similar to the previous example, but neither line of the constructor will compile since they are using the keywords incorrectly. When you see `super()` or `super` on the exam, be sure to check that they are being used correctly.

Inheriting Methods

Inheriting a class grants us access to the `public` and `protected` members of the parent class, but also sets the stage for collisions between methods defined in both the parent class and the subclass. In this section, we'll review the rules for method inheritance and how Java handles such scenarios.

Overriding a Method

What if there is a method defined in both the parent and child class? For example, you may want to define a new version of an existing method in a child class that makes use of the definition in the parent class. In this case, you can *override* a method a method by declaring a new method with the signature and return type as the method in the parent class. As you may recall from Chapter 4, the method signature includes the name and list of input parameters.

When you override a method, you may reference the parent version of the method using the super keyword. In this manner, the keywords this and super allow you to select between the current and parent version of a method, respectively. We illustrate this with the following example:

```
public class Canine {
  public double getAverageWeight() {
    return 50;
  }
}

public class Wolf extends Canine {
  public double getAverageWeight() {
    return super.getAverageWeight()+20;
  }
  public static void main(String[] args) {
    System.out.println(new Canine().getAverageWeight());
    System.out.println(new Wolf().getAverageWeight());
  }
}
```

In this example, in which the child class `Wolf` overrides the parent class `Canine`, the method `getAverageWeight()` runs without issue and outputs the following:

```
50.00
70.00
```

You might be wondering, was the use of `super` in the child's method required? For example, what would the following code output if we removed the `super` keyword in the `getAverageWeight()` method of the `Wolf` class?

```
public double getAverageWeight() {
  return getAverageWeight()+20;  // INFINITE LOOP
}
```

In this example, the compiler would not call the parent `Canine` method; it would call the current `Wolf` method since it would think you were executing a recursive call. A recursive function is one that calls itself as part of execution, and it is common in programming. A recursive function must have a termination condition. In this example, there is no termination condition; therefore, the application will attempt to call itself infinitely and produce a stack overflow error at runtime.

Overriding a method is not without limitations, though. The compiler performs the following checks when you override a nonprivate method:

1. The method in the child class must have the same signature as the method in the parent class.

2. The method in the child class must be at least as accessible or more accessible than the method in the parent class.
3. The method in the child class may not throw a checked exception that is new or broader than the class of any exception thrown in the parent class method.
4. If the method returns a value, it must be the same or a subclass of the method in the parent class, known as *covariant return types*.

The first rule of overriding a method is somewhat self-explanatory. If two methods have the same name but different signatures, the methods are overloaded, not overridden. As you may recall from our discussion of overloaded methods in Chapter 4, the methods are unrelated to each other and do not share any properties.

Overloading vs. Overriding

Overloading a method and overriding a method are similar in that they both involve redefining a method using the same name. They differ in that an overloaded method will use a different signature than an overridden method. This distinction allows overloaded methods a great deal more freedom in syntax than an overridden method would have. For example, take a look at the following code sample:

```
public class Bird {
  public void fly() {
    System.out.println("Bird is flying");
  }
  public void eat(int food) {
    System.out.println("Bird is eating "+food+" units of food");
  }
}

public class Eagle extends Bird {
  public int fly(int height) {
    System.out.println("Bird is flying at "+height+" meters");
    return height;
  }
  public int eat(int food) { // DOES NOT COMPILE
    System.out.println("Bird is eating "+food+" units of food");
    return food;
  }
}
```

The first method, `fly()`, is overloaded in the subclass `Eagle`, since the signature changes from a no-argument constructor to a constructor with one `int` argument. Because the method is being overloaded and not overridden, the return type can be changed from `void` to `int` without issue.

The second method, `eat()`, is overridden in the subclass `Eagle`, since the signature is the same as it is in the parent class `Bird`—they both take a single argument `int`. Because the method is being overridden, the return type of the method in `Eagle` must be a subclass of the return type of the method in `Bird`. In this example, the return type `void` is not a subclass of `int`; therefore, the compiler will throw an exception on this method definition.

Any time you see a method on the exam with the same name as a method in the parent class, determine whether the method is being overloaded or overridden first; doing so will help you with questions about whether the code will compile.

Let's review some examples of the last three rules of overriding methods so you can learn to spot the issues when they arise:

```
public class Camel {
  protected String getNumberOfHumps() {
    return "Undefined";
  }
}

public class BactrianCamel extends Camel {
  private int getNumberOfHumps() {  // DOES NOT COMPILE
    return 2;
  }
}
```

In this example, the method in the child class doesn't compile for two reasons. First, it violates the second rule of overriding methods: the child method must be at least as accessible as the parent. In the example, the parent method uses the `protected` modifier, but the child method uses the `private` modifier, making it less accessible in the child method than in the parent method. It also violates the fourth rule of overriding methods: the return type of the parent method and child method must be covariant. In this example, the return type of the parent method is `String`, whereas the return type of the child method is `int`, neither of which is covariant with each other.

Any time you see a method that appears to be overridden on the example, first check to make sure it is truly being overridden and not overloaded. Once you have confirmed it is being overridden, check that the access modifiers, return types, and any exceptions defined in the method are compatible with one another. Let's take a look at some example methods that use exceptions:

```
public class InsufficientDataException extends Exception {}

public class Reptile {
  protected boolean hasLegs() throws InsufficientDataException {
    throw new InsufficientDataException();
  }
  protected double getWeight() throws Exception {
    return 2;
  }
}

public class Snake extends Reptile {
  protected boolean hasLegs() {
    return false;
  }
  protected double getWeight() throws InsufficientDataException{
    return 2;
  }
}
```

In this example, both parent and child classes define two methods, `hasLegs()` and `getWeight()`. The first method, `hasLegs()`, throws an exception `InsufficientDataException` in the parent class but doesn't throw an exception in the child class. This does not violate the third rule of overriding methods, though, as no new exception is defined. In other words, a child method may hide or eliminate a parent method's exception without issue.

The second method, `getWeight()`, throws `Exception` in the parent class and `InsufficientDataException` in the child class. This is also permitted, as `InsufficientDataException` is a subclass of `Exception` by construction.

Neither of the methods in the previous example violates the third rule of overriding methods, so the code compiles and runs without issue. Let's review some examples that do violate the third rule of overriding methods:

```
public class InsufficientDataException extends Exception {}

public class Reptile {
  protected double getHeight() throws InsufficientDataException {
    return 2;
  }
  protected int getLength() {
    return 10;
  }
}
```

```
public class Snake extends Reptile {
  protected double getHeight() throws Exception {  // DOES NOT COMPILE
    return 2;
  }
  protected int getLength() throws InsufficientDataException { // DOES NOT COMPILE
    return 10;
  }
}
```

Unlike the earlier example, neither of the methods in the child class of this code will compile. The `getHeight()` method in the parent class throws an `InsufficientDataException`, whereas the method in the child class throws an `Exception`. Since `Exception` is not a subclass of `InsufficientDataException`, the third rule of overriding methods is violated and the code will not compile. Coincidentally, `Exception` is a superclass of `InsufficientDataException`.

Next, the `getLength()` method doesn't throw an exception in the parent class, but it does throw an exception, `InsufficientDataException`, in the child class. In this manner, the child class defines a new exception that the parent class did not, which is a violation of the third rule of overriding methods.

The last three rules of overriding a method may seem arbitrary or confusing at first, but as you'll see later in this chapter when we discuss polymorphism, they are needed for consistency of the language. Without these rules in place, it is possible to create contradictions within the Java language.

Redeclaring *private* Methods

The previous section defined the behavior if you override a `public` or `protected` method in the class. Now let's expand our discussion to `private` methods. In Java, it is not possible to override a `private` method in a parent class since the parent method is not accessible from the child class. Just because a child class doesn't have access to the parent method, doesn't mean the child class can't define its own version of the method. It just means, strictly speaking, that the new method is not an overridden version of the parent class's method.

Java permits you to redeclare a new method in the child class with the same or modified signature as the method in the parent class. This method in the child class is a separate and independent method, unrelated to the parent version's method, so none of the rules for overriding methods are invoked. For example, let's return to the `Camel` example we used in the previous section and show two related classes that define the same method:

```
public class Camel {
  private String getNumberOfHumps() {
    return "Undefined";
  }
}
```

```
public class BactrianCamel extends Camel {
  private int getNumberOfHumps() {
    return 2;
  }
}
```

This code compiles without issue. Notice that the return type differs in the child method from `String` to `int`. In this example, the method `getNumberOfHumps()` in the parent class is hidden, so the method in the child class is a new method and not an override of the method in the parent class. As you saw in the previous section, if the method in the parent class were `public` or `protected`, the method in the child class would not compile because it would violate two rules of overriding methods. The parent method in this example is `private`, so there are no such issues.

Hiding Static Methods

A *hidden method* occurs when a child class defines a static method with the same name and signature as a static method defined in a parent class. Method hiding is similar but not exactly the same as method overriding. First, the four previous rules for overriding a method must be followed when a method is hidden. In addition, a new rule is added for hiding a method, namely that the usage of the `static` keyword must be the same between parent and child classes. The following list summarizes the five rules for hiding a method:

1. The method in the child class must have the same signature as the method in the parent class.
2. The method in the child class must be at least as accessible or more accessible than the method in the parent class.
3. The method in the child class may not throw a checked exception that is new or broader than the class of any exception thrown in the parent class method.
4. If the method returns a value, it must be the same or a subclass of the method in the parent class, known as *covariant return types*.
5. The method defined in the child class must be marked as `static` if it is marked as `static` in the parent class (method hiding). Likewise, the method must not be marked as `static` in the child class if it is not marked as `static` in the parent class (method overriding).

Note that the first four are the same as the rules for overriding a method.

Let's review some examples of the new rule:

```
public class Bear {
  public static void eat() {
    System.out.println("Bear is eating");
  }
}
```

```
public class Panda extends Bear {
  public static void eat() {
    System.out.println("Panda bear is chewing");
  }
  public static void main(String[] args) {
    Panda.eat();
  }
}
```

In this example, the code compiles and runs without issue. The `eat()` method in the child class hides the `eat()` method in the parent class. Because they are both marked as `static`, this is not considered an overridden method. Let's contrast this with examples that violate the fifth rule:

```
public class Bear {
  public static void sneeze() {
    System.out.println("Bear is sneezing");
  }
  public void hibernate() {
    System.out.println("Bear is hibernating");
  }
}

public class Panda extends Bear {
  public void sneeze() {  // DOES NOT COMPILE
    System.out.println("Panda bear sneezes quietly");
  }
  public static void hibernate() {  // DOES NOT COMPILE
    System.out.println("Panda bear is going to sleep");
  }
}
```

In this example, `sneeze()` is marked as `static` in the parent class but not in the child class. The compiler detects that you're trying to override a method that should be hidden and generates a compiler error. In the second method, `hibernate()` is an instance member in the parent class but a static method in the child class. In this scenario, the compiler thinks that you're trying to hide a method that should be overridden and also generates a compiler error.

As you saw in the previous example, hiding static methods is fraught with pitfalls and potential problems and as a practice should be avoided. Though you might see questions on the exam that contain hidden static methods that are syntactically correct, avoid hiding static methods in your own work, since it tends to lead to confusing and difficult-to-read code. You should not reuse the name of a static method in your class if it is already used in the parent class.

Overriding vs. Hiding Methods

In our description of hiding of static methods, we indicated there was a distinction between overriding and hiding methods. Unlike overriding a method, in which a child method replaces the parent method in calls defined in both the parent and child, hidden methods only replace parent methods in the calls defined in the child class.

At runtime the child version of an overridden method is always executed for an instance regardless of whether the method call is defined in a parent or child class method. In this manner, the parent method is never used unless an explicit call to the parent method is referenced, using the syntax *ParentClassName.method()*. Alternatively, at runtime the parent version of a hidden method is always executed if the call to the method is defined in the parent class. Let's take a look at an example:

```
public class Marsupial {
  public static boolean isBiped() {
    return false;
  }
  public void getMarsupialDescription() {
    System.out.println("Marsupial walks on two legs: "+isBiped());
  }
}

public class Kangaroo extends Marsupial {
  public static boolean isBiped() {
    return true;
  }
  public void getKangarooDescription() {
    System.out.println("Kangaroo hops on two legs: "+isBiped());
  }
  public static void main(String[] args) {
    Kangaroo joey = new Kangaroo();
    joey.getMarsupialDescription();
    joey.getKangarooDescription();
  }
}
```

In this example, the code compiles and runs without issue, outputting the following:

```
Marsupial walks on two legs: false
Kangaroo hops on two legs: true
```

Notice that `isBiped()` returns `false` in the parent class and `true` in the child class. In the first method call, the parent method `getMarsupialDescription()` is used. The `Marsupial` class only knows about `isBiped()` from its own class definition, so it outputs `false`. In the second method call, the child executes a method of `isBiped()`, which hides the parent method's version and returns `true`.

Contrast this first example with the following example, which uses an overridden version of `isBiped()` instead of a hidden version:

```
class Marsupial {
  public boolean isBiped() {
    return false;
  }
  public void getMarsupialDescription() {
    System.out.println("Marsupial walks on two legs: "+isBiped());
  }
}

public class Kangaroo extends Marsupial {
  public boolean isBiped() {
    return true;
  }
  public void getKangarooDescription() {
    System.out.println("Kangaroo hops on two legs: "+isBiped());
  }
  public static void main(String[] args) {
    Kangaroo joey = new Kangaroo();
    joey.getMarsupialDescription();
    joey.getKangarooDescription();
  }
}
```

This code also compiles and runs without issue, but it outputs slightly different text:

```
Marsupial walks on two legs: true
Kangaroo hops on two legs: true
```

In this example, the `isBiped()` method is overridden, not hidden, in the child class. Therefore, it is replaced at runtime in the parent class with the call to the child class's method.

Make sure you understand these examples as they show how hidden and overridden methods are fundamentally different. This example makes uses of polymorphism, which we'll discuss later in this chapter.

Creating *final* methods

We conclude our discussion of method inheritance with a somewhat self-explanatory rule: `final` methods cannot be overridden. If you recall our discussion of modifiers from Chapter 4, you can create a method with the `final` keyword. By doing so, though, you forbid a child class from overriding this method. This rule is in place both when you override a method and when you hide a method. In other words, you cannot hide a static method in a parent class if it is marked as `final`.

Let's take a look at an example:

```
public class Bird {
  public final boolean hasFeathers() {
    return true;
  }
}

public class Penguin extends Bird {
  public final boolean hasFeathers() { // DOES NOT COMPILE
    return false;
  }
}
```

In this example, the method `hasFeathers()` is marked as `final` in the parent class `Bird`, so the child class `Penguin` cannot override the parent method, resulting in a compiler error. Note that whether or not the child method used the `final` keyword is irrelevant—the code will not compile either way.

Why Mark a Method as *final?*

Although marking methods as `final` prevents them from being overridden, it does have advantages in practice. For example, you'd mark a method as `final` when you're defining a parent class and want to guarantee certain behavior of a method in the parent class, regardless of which child is invoking the method.

For example, in the previous example with `Birds`, the author of the parent class may want to ensure the method `hasFeathers()` always returns `true`, regardless of the child class instance on which it is invoked. The author is confident that there is no example of a `Bird` in which feathers are not present.

The reason methods are not commonly marked as `final` in practice, though, is that it may be difficult for the author of a parent class method to consider all of the possible ways her child class may be used. For example, although all adult birds have feathers, a baby chick doesn't; therefore, if you have an instance of a `Bird` that is a chick, it would not have feathers. In short, the `final` modifier is only used on methods when the author of the parent method wants to guarantee very precise behavior.

Inheriting Variables

As you saw with method overriding, there were a lot of rules when two methods have the same signature and are defined in both the parent and child classes. Luckily, the rules for variables with the same name in the parent and child classes are a lot simpler, because Java doesn't allow variables to be overridden but instead hidden.

Hiding Variables

When you hide a variable, you define a variable with the same name as a variable in a parent class. This creates two copies of the variable within an instance of the child class: one instance defined for the parent reference and another defined for the child reference.

As when hiding a static method, you can't override a variable; you can only hide it. Also similar to hiding a static method, the rules for accessing the parent and child variables are quite similar. If you're referencing the variable from within the parent class, the variable defined in the parent class is used. Alternatively, if you're referencing the variable from within a child class, the variable defined in the child class is used. Likewise, you can reference the parent value of the variable with an explicit use of the `super` keyword. Consider the following example:

```
public class Rodent {
  protected int tailLength = 4;
  public void getRodentDetails() {
    System.out.println("[parentTail="+tailLength+"]");
  }
}

public class Mouse extends Rodent {
  protected int tailLength = 8;
  public void getMouseDetails() {
    System.out.println("[tail="+tailLength +",parentTail="+super.tailLength+"]");
  }
  public static void main(String[] args) {
    Mouse mouse = new Mouse();
    mouse.getRodentDetails();
    mouse.getMouseDetails();
  }
}
```

This code compiles without issue and outputs the following when executed:

```
[parentTail=4]
[tail=8,parentTail=4]
```

Notice that the instance of `Mouse` contains two copies of the `tailLength` variables: one defined in the parent and one defined in the child. These instances are kept separate from each other, allowing our instance of `Mouse` to reference both `tailLength` values independently. In the first method call, `getRodentDetails()`, the parent method outputs the parent value of the `tailLength` variable. In the second method call, `getMouseDetails()`, the child method outputs both the child and parent version of the `tailLength` variables, using the `super` keyword to access the parent variable's value.

The important thing to remember is that there is no notion of overriding a member variable. For example, there is no code change that could be made to cause Java to override the value of `tailLength`, making it the same in both parent and child. These rules are the same regardless of whether the variable is an instance variable or a static variable.

Real World Scenario

Don't Hide Variables in Practice

Although Java allows you to hide a variable defined in a parent class with one defined in a child class, it is considered an extremely poor coding practice. For example, take a look at the following code, which uses a hidden variable `length`, marked as `public` in both parent and child classes.

```
public class Animal {
  public int length = 2;
}

public class Jellyfish extends Animal {
  public int length = 5;
  public static void main(String[] args) {
    Jellyfish jellyfish = new Jellyfish();
    Animal animal = new Jellyfish();
    System.out.println(jellyfish.length);
    System.out.println(animal.length);
  }
}
```

This code compiles without issue. Here's the output:

```
5
2
```

Notice the same type of object was created twice, but the reference to the object determines which value is seen as output. If the object `Jellyfish` was passed to a method by an `Animal` reference, as you'll see in the section "Understanding Polymorphism," later in this chapter, the wrong value might be used.

Hiding variables makes the code very confusing and difficult to read, especially if you start modifying the value of the variable in both the parent and child methods, since it may not be clear which variable you're updating.

When defining a new variable in a child class, it is considered good coding practice to select a name for the variable that is not already a `public`, `protected`, or default variable in use in a parent class. Hiding `private` variables is considered less problematic because the child class did not have access to the variable in the parent class to begin with.

Creating Abstract Classes

Let's say you want to define a parent class that other developers are going to subclass. Your goal is to provide some reusable variables and methods to developers in the parent class, whereas the developers provide specific implementations or overrides of other methods in the child classes. Furthermore, let's say you also don't want an instance of the parent class to be instantiated unless it is an instance of the child class.

For example, you might define an `Animal` parent class that a number of classes extend from and use but for which an instance of `Animal` itself cannot be instantiated. All subclasses of the `Animal` class, such as `Swan`, are required to implement a `getName()` method, but there is no implementation for the method in the parent `Animal` class. How do you ensure all classes that extend `Animal` provide an implementation for this method?

In Java, you can accomplish this task by using an abstract class and abstract method. An *abstract class* is a class that is marked with the `abstract` keyword and cannot be instantiated. An *abstract method* is a method marked with the `abstract` keyword defined in an abstract class, for which no implementation is provided in the class in which it is declared.

The following code is based on our `Animal` and `Swan` description:

```
public abstract class Animal {
  protected int age;
  public void eat() {
    System.out.println("Animal is eating");
  }
  public abstract String getName();
}
```

```
public class Swan extends Animal {
  public String getName() {
    return "Swan";
  }
}
```

The first thing to notice about this sample code is that the `Animal` class is declared abstract and `Swan` is not. Next, the member `age` and the method `eat()` are marked as `protected` and `public`, respectively; therefore, they are inherited in subclasses such as `Swan`. Finally, the abstract method `getName()` is terminated with a semicolon and doesn't provide a body in the parent class `Animal`. This method is implemented with the same name and signature as the parent method in the `Swan` class.

Defining an Abstract Class

The previous sample code illustrates a number of important rules about abstract classes. For example, an abstract class may include nonabstract methods and variables, as you saw with the variable `age` and the method `eat()`. In fact, an abstract class is not required to include any abstract methods. For example, the following code compiles without issue even though it doesn't define any abstract methods:

```
public abstract class Cow {
}
```

Although an abstract class doesn't have to implement any abstract methods, an abstract method may only be defined in an abstract class. For example, the following code won't compile because an abstract method is not defined within an abstract class:

```
public class Chicken {
  public abstract void peck();  // DOES NOT COMPILE
}
```

The exam creators are fond of questions like this one, which mixes nonabstract classes with abstract methods. They are also fond of questions with methods marked as `abstract` for which an implementation is also defined. For example, neither method in the following code will compile because the methods are marked as `abstract`:

```
public abstract class Turtle {
  public abstract void swim() {}  // DOES NOT COMPILE
  public abstract int getAge() {  // DOES NOT COMPILE
    return 10;
  }
}
```

The first method, `swim()`, doesn't compile because two brackets are provided instead of a semicolon, and Java interprets this as providing a body to an abstract method. The second method, `getAge()`, doesn't compile because it also provides a body to an abstract method. Pay close attention to `swim()`, because you'll likely see a question like this on the exam.

Default Method Implementations in Abstract Classes

Although you can't provide a default implementation to an abstract method in an abstract class, you can still define a method with a body—you just can't mark it as `abstract`. As long as you do not mark it as `final`, the subclass still has the option to override it, as explained in the previous section.

Next, we note that an abstract class cannot be marked as `final` for a somewhat obvious reason. By definition, an abstract class is one that must be extended by another class to be instantiated, whereas a `final` class can't be extended by another class. By marking an abstract class as `final`, you're saying the class can never be instantiated, so the compiler refuses to process the code. For example, the following code snippet will not compile:

```
public final abstract class Tortoise {  // DOES NOT COMPILE
}
```

Likewise, an abstract method may not be marked as `final` for the same reason that an abstract class may not be marked as `final`. Once marked as `final`, the method can never be overridden in a subclass, making it impossible to create a concrete instance of the abstract class.

```
public abstract class Goat {
  public abstract final void chew();  // DOES NOT COMPILE
}
```

Finally, a method may not be marked as both `abstract` and `private`. This rule makes sense if you think about it. How would you define a subclass that implements a required method if the method is not accessible by the subclass itself? The answer is you can't, which is why the compiler will complain if you try to do the following:

```
public abstract class Whale {
  private abstract void sing();  // DOES NOT COMPILE
}

public class HumpbackWhale extends Whale {
  private void sing() {
    System.out.println("Humpback whale is singing");
  }
}
```

In this example, the abstract method `sing()` defined in the parent class `Whale` is not visible to the subclass `HumpbackWhale`. Even though `HumpbackWhale` does provide an implementation, it is not considered an override of the abstract method since the abstract method is unreachable. The compiler recognizes this in the parent class and throws an exception as soon as `private` and `abstract` are applied to the same method.

If we changed the access modified from `private` to `protected` in the parent class `Whale`, would the code compile? Let's take a look:

```
public abstract class Whale {
  protected abstract void sing();
}

public class HumpbackWhale extends Whale {
  private void sing() {  // DOES NOT COMPILE
    System.out.println("Humpback whale is singing");
  }
}
```

In this modified example, the code will still not compile but for a completely different reason. If you remember the rules earlier in this chapter for overriding a method, the subclass cannot reduce the visibility of the parent method, `sing()`. Because the method is declared `protected` in the parent class, it must be marked as `protected` or `public` in the child class. Even with abstract methods, the rules for overriding methods must be followed.

Creating a Concrete Class

When working with abstract classes, it is important to remember that by themselves, they cannot be instantiated and therefore do not do much other than define static variables and methods. For example, the following code will not compile as it is an attempt to instantiate an abstract class.

```
public abstract class Eel {
  public static void main(String[] args) {
    final Eel eel = new Eel();  // DOES NOT COMPILE
  }
}
```

An abstract class becomes useful when it is extended by a concrete subclass. A *concrete class* is the first nonabstract subclass that extends an abstract class and is required to implement all inherited abstract methods. When you see a concrete class extending an abstract class on the exam, check that it implements all of the required abstract methods. Let's review this with the following example.

```
public abstract class Animal {
  public abstract String getName();
}
```

```
public class Walrus extends Animal { // DOES NOT COMPILE
}
```

First, note that `Animal` is marked as `abstract` and `Walrus` is not. In this example, `Walrus` is considered the first concrete subclass of `Animal`. Second, since `Walrus` is the first concrete subclass, it must implement all inherited abstract methods, `getName()` in this example. Because it doesn't, the compiler rejects the code.

Notice that when we define a concrete class as the "first" nonabstract subclass, we include the possibility that another nonabstract class may extend an existing nonabstract class. The key point is that the first class to extend the nonabstract class must implement all inherited abstract methods. For example, the following variation will also not compile:

```
public abstract class Animal {
  public abstract String getName();
}

public class Bird extends Animal { // DOES NOT COMPILE
}

public class Flamingo extends Bird {
  public String getName() {
    return "Flamingo";
  }
}
```

Even though a second subclass `Flamingo` implements the abstract method `getName()`, the first concrete subclass `Bird` doesn't; therefore, the `Bird` class will not compile.

Extending an Abstract Class

Let's expand our discussion of abstract classes by introducing the concept of extending an abstract class with another abstract. We'll repeat our previous `Walrus` example with one minor variation:

```
public abstract class Animal {
  public abstract String getName();
}

public class Walrus extends Animal { // DOES NOT COMPILE
}

public abstract class Eagle extends Animal {
}
```

In this example, we again have an abstract class `Animal` with a concrete subclass `Walrus` that doesn't compile since it doesn't implement a `getName()` method. We also have an abstract class `Eagle`, which like `Walrus` extends `Animal` and doesn't provide an implementation for `getName()`. In this situation, `Eagle` does compile because it is marked as `abstract`. Be sure you understand why `Walrus` doesn't compile and `Eagle` does in this example.

As you saw in this example, abstract classes can extend other abstract classes and are not required to provide implementations for any of the abstract methods. It follows, then, that a concrete class that `extends` an abstract class must implement all inherited abstract methods. For example, the following concrete class `Lion` must implement two methods, `getName()` and `roar()`:

```
public abstract class Animal {
  public abstract String getName();
}

public abstract class BigCat extends Animal {
  public abstract void roar();
}

public class Lion extends BigCat {
  public String getName() {
    return "Lion";
  }
  public void roar() {
    System.out.println("The Lion lets out a loud ROAR!");
  }
}
```

In this sample code, `BigCat` extends `Animal` but is marked as `abstract`; therefore, it is not required to provide an implementation for the `getName()` method. The class `Lion` is not marked as `abstract`, and as the first concrete subclass, it must implement all inherited abstract methods not defined in a parent class.

There is one exception to the rule for abstract methods and concrete classes: a concrete subclass is not required to provide an implementation for an abstract method if an intermediate abstract class provides the implementation. For example, take a look at the following variation on our previous example:

```
public abstract class Animal {
  public abstract String getName();
}

public abstract class BigCat extends Animal {
  public String getName() {
```

```
    return "BigCat";
  }
  public abstract void roar();
}

public class Lion extends BigCat {
  public void roar() {
    System.out.println("The Lion lets out a loud ROAR!");
  }
}
```

In this example, `BigCat` provides an implementation for the abstract method `getName()` defined in the abstract `Animal` class. Therefore, `Lion` inherits only one abstract method, `roar()`, and is not required to provide an implementation for the method `getName()`.

Here's one way to think about this: if an intermediate class provides an implementation for an abstract method, that method is inherited by subclasses as a concrete method, not as an abstract one. In other words, the subclasses do not consider it an inherited abstract method because it is no longer abstract by the time it reaches the subclasses.

The following are lists of rules for abstract classes and abstract methods that we have covered in this section. Review and understand these rules before taking the exam.

Abstract Class Definition Rules:

1. Abstract classes cannot be instantiated directly.
2. Abstract classes may be defined with any number, including zero, of abstract and non-abstract methods.
3. Abstract classes may not be marked as `private` or `final`.
4. An abstract class that `extends` another abstract class inherits all of its abstract methods as its own abstract methods.
5. The first concrete class that `extends` an abstract class must provide an implementation for all of the inherited abstract methods.

Abstract Method Definition Rules:

1. Abstract methods may only be defined in abstract classes.
2. Abstract methods may not be declared `private` or `final`.
3. Abstract methods must not provide a method body/implementation in the abstract class for which is it declared.
4. Implementing an abstract method in a subclass follows the same rules for overriding a method. For example, the name and signature must be the same, and the visibility of the method in the subclass must be at least as accessible as the method in the parent class.

Implementing Interfaces

Although Java doesn't allow multiple inheritance, it does allow classes to implement any number of interfaces. An *interface* is an abstract data type that defines a list of abstract public methods that any class implementing the interface must provide. An interface can also include a list of constant variables and default methods, which we'll cover in this section. In Java, an interface is defined with the `interface` keyword, analogous to the `class` keyword used when defining a class. A class invokes the interface by using the `implements` keyword in its class definition. Refer to Figures 5.4 and 5.5 for proper syntax usage.

FIGURE 5.4 Defining an interface

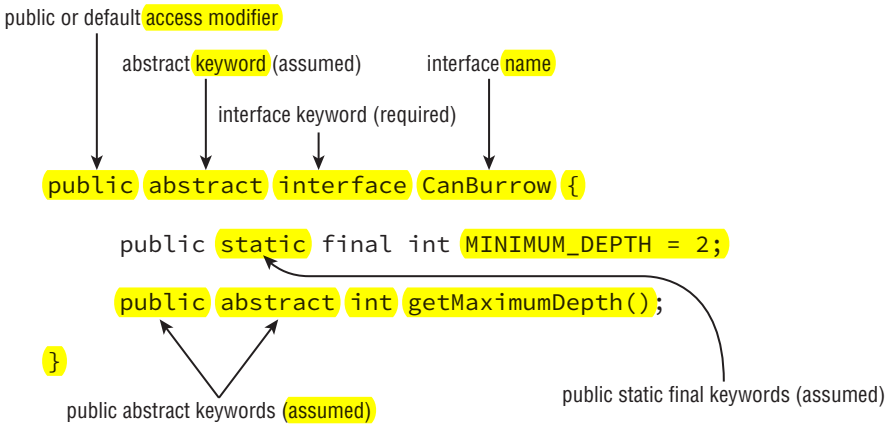
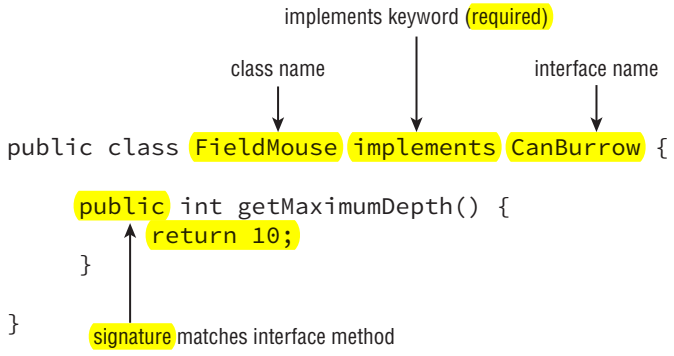


FIGURE 5.5 Implementing an interface



As you see in this example, an interface is not declared an abstract class, although it has many of the same properties of abstract class. Notice that the method modifiers in this

example, `abstract` and `public`, are assumed. In other words, whether or not you provide them, the compiler will automatically insert them as part of the method definition.

A class may implement multiple interfaces, each separated by a comma, such as in the following example:

```
public class Elephant implements WalksOnFourLegs, HasTrunk, Herbivore {
}
```

In the example, if any of the interfaces defined abstract methods, the concrete class `Elephant` would be required to implement those methods.

New to Java 8 is the notion of default and static interface methods, which we'll cover at the end of this section.

Defining an Interface

It may be helpful to think of an interface as a specialized kind of abstract class, since it shares many of the same properties and rules as an abstract class. The following is a list of rules for creating an interface, many of which you should recognize as adaptions of the rules for defining abstract classes.

1. Interfaces cannot be instantiated directly.
2. An interface is not required to have any methods.
3. An interface may not be marked as `final`.
4. All top-level interfaces are assumed to have `public` or default access, and they must include the `abstract` modifier in their definition. Therefore, marking an interface as `private`, `protected`, or `final` will trigger a compiler error, since this is incompatible with these assumptions.
5. All nondefault methods in an interface are assumed to have the modifiers `abstract` and `public` in their definition. Therefore, marking a method as `private`, `protected`, or `final` will trigger compiler errors as these are incompatible with the `abstract` and `public` keywords.

The fourth rule doesn't apply to inner interfaces, although inner classes and interfaces are not in scope for the OCA exam. The first three rules are identical to the first three rules for creating an abstract class. Imagine we have an interface `WalksOnTwoLegs`, defined as follows:

```
public interface WalksOnTwoLegs {}
```

It compiles without issue, since interfaces are not required to define any methods. Now consider the following two examples, which do not compile:

```
public class TestClass {
  public static void main(String[] args) {
    WalksOnTwoLegs example = new WalksOnTwoLegs();  // DOES NOT COMPILE
```

```
  }
}

public final interface WalksOnEightLegs {  // DOES NOT COMPILE
}
```

The first example doesn't compile, as `WalksOnTwoLegs` is an interface and cannot be instantiated directly. The second example, `WalksOnEightLegs`, doesn't compile since interfaces may not be marked as `final` for the same reason that `abstract` classes cannot be marked as `final`.

The fourth and fifth rule about "assumed keywords" might be new to you, but you should think of these in the same light as the compiler inserting a default no-argument constructor or `super()` statement into your constructor. You may provide these modifiers yourself, although the compiler will insert them automatically if you do not. For example, the following two interface definitions are equivalent, as the compiler will convert them both to the second example:

```
public interface CanFly {
  void fly(int speed);
  abstract void takeoff();
  public abstract double dive();
}

public abstract interface CanFly {
  public abstract void fly(int speed);
  public abstract void takeoff();
  public abstract double dive();
}
```

In this example, the `abstract` keyword is first automatically added to the interface definition. Then, each method is prepended with `abstract` and `public` keywords. If the method already has either of these keywords, then no change is required. Let's take a look at an example that violates the assumed keywords:

```
private final interface CanCrawl {  // DOES NOT COMPILE
  private void dig(int depth);  // DOES NOT COMPILE
  protected abstract double depth();  // DOES NOT COMPILE
  public final void surface();  // DOES NOT COMPILE
}
```

Every single line of this example doesn't compile. The first line doesn't compile for two reasons. First, it is marked as `final`, which cannot be applied to an interface since it conflicts with the assumed `abstract` keyword. Next, it is marked as `private`, which conflicts with the `public` or default required access for interfaces. The second and third line do not compile because all interface methods are assumed to be `public` and marking them

as `private` or `protected` throws a compiler error. Finally, the last line doesn't compile because the method is marked as `final` and since interface methods are assumed to be `abstract`, the compiler throws an exception for using both `abstract` and `final` keywords on a method.

Adding the assumed keywords to an interface is a matter of personal preference, although it is considered good coding practice to do so. Code with the assumed keywords written out tends to be easier and clearer to read, and leads to fewer potential conflicts, as you saw in the previous examples.

Be sure to review the previous example and understand why each of the lines doesn't compile. There will likely be at least one question on the exam in which an interface or interface method uses an invalid modifier.

Inheriting an Interface

There are two inheritance rules you should keep in mind when extending an interface:

1. An interface that `extends` another interface, as well as an abstract class that `implements` an interface, inherits all of the abstract methods as its own abstract methods.
2. The first concrete class that `implements` an interface, or `extends` an abstract class that `implements` an interface, must provide an implementation for all of the inherited abstract methods.

Like an abstract class, an interface may be extended using the `extend` keyword. In this manner, the new child interface inherits all the abstract methods of the parent interface. Unlike an abstract class, though, an interface may extend multiple interfaces. Consider the following example:

```
public interface HasTail {
  public int getTailLength();
}

public interface HasWhiskers {
  public int getNumberOfWhiskers();
}

public interface Seal extends HasTail, HasWhiskers {
}
```

Any class that implements the `Seal` interface must provide an implementation for all methods in the parent interfaces—in this case, `getTailLength()` and `getNumberOfWhiskers()`.

What about an abstract class that implements an interface? In this scenario, the abstract class is treated in the same way as an interface extending another interface. In other words, the abstract class inherits the abstract methods of the interface but is not required to implement them. That said, like an abstract class, the first concrete class to extend the abstract class must implement all the inherited abstract methods of the interface. We illustrate this in the following example:

```
public interface HasTail {
  public int getTailLength();
}

public interface HasWhiskers {
  public int getNumberOfWhiskers();
}

public abstract class HarborSeal implements HasTail, HasWhiskers {
}

public class LeopardSeal implements HasTail, HasWhiskers {  // DOES NOT COMPILE
}
```

In this example, we see that `HarborSeal` is an abstract class and compiles without issue. Any class that `extends HarborSeal` will be required to implement all of the methods in the `HasTail` and `HasWhiskers` interface. Alternatively, `LeopardSeal` is not an abstract class, so it must implement all the interface methods within its definition. In this example, `LeopardSeal` doesn't provide an implementation for the interface methods, so the code doesn't compile.

Classes, Interfaces, and Keywords

The exam creators are fond of questions that mix class and interface terminology. Although a class can implement an interface, a class cannot extend an interface. Likewise, whereas an interface can extend another interface, an interface cannot implement another interface. The following examples illustrate these principles:

```
public interface CanRun {}

public class Cheetah extends CanRun {}  // DOES NOT COMPILE

public class Hyena {}

public interface HasFur extends Hyena {} // DOES NOT COMPILE
```

The first example shows a class trying to extend an interface that doesn't compile. The second example shows an interface trying to extend a class, which also doesn't compile.

Be wary of examples on the exam that mix class and interface definitions. Make sure the only connection between a class and an interface is with the *class* `implements` *interface* syntax.

Abstract Methods and Multiple Inheritance

Since Java allows for multiple inheritance via interfaces, you might be wondering what will happen if you define a class that inherits from two interfaces that contain the same `abstract` method:

```
public interface Herbivore {
  public void eatPlants();
}

public interface Omnivore {
  public void eatPlants();
  public void eatMeat();
}
```

In this scenario, the signatures for the two interface methods `eatPlants()` are compatible, so you can define a class that fulfills both interfaces simultaneously:

```
public class Bear implements Herbivore, Omnivore {
  public void eatMeat() {
    System.out.println("Eating meat");
  }
  public void eatPlants() {
    System.out.println("Eating plants");
  }
}
```

Why does this work? Remember that interface methods in this example are `abstract` and define the "behavior" that the class implementing the interface must have. If two `abstract` interface methods have identical behaviors—or in this case the same method signature— creating a class that implements one of the two methods automatically implements the second method. In this manner, the interface methods are considered duplicates since they have the same signature.

What happens if the two methods have different signatures? If the method name is the same but the input parameters are different, there is no conflict because this is considered a method overload. We demonstrate this principle in the following example:

```
public interface Herbivore {
  public int eatPlants(int quantity);
}
```

```
public interface Omnivore {
  public void eatPlants();
}

public class Bear implements Herbivore, Omnivore {
  public int eatPlants(int quantity) {
    System.out.println("Eating plants: "+quantity);
    return quantity;
  }
  public void eatPlants() {
    System.out.println("Eating plants");
  }
}
```

In this example, we see that the class that implements both interfaces must provide implements of both versions of `eatPlants()`, since they are considered separate methods. Notice that it doesn't matter if the return type of the two methods is the same or different, because the compiler treats these methods as independent.

Unfortunately, if the method name and input parameters are the same but the return types are different between the two methods, the class or interface attempting to inherit both interfaces will not compile. The reason the code doesn't compile has less to do with interfaces and more to do with class design, as discussed in Chapter 4. It is not possible in Java to define two methods in a class with the same name and input parameters but different return types. Given the following two interface definitions for `Herbivore` and `Omnivore`, the following code will not compile:

```
public interface Herbivore {
  public int eatPlants();
}

public interface Omnivore {
  public void eatPlants();
}

public class Bear implements Herbivore, Omnivore {
  public int eatPlants() {  // DOES NOT COMPILE
    System.out.println("Eating plants: 10");
    return 10;
  }
  public void eatPlants() {  // DOES NOT COMPILE
    System.out.println("Eating plants");
  }
}
```

The code doesn't compile, as the class defines two methods with the same name and input parameters but different return types. If we were to remove either definition of `eatPlants()`, the compiler would stop because the definition of `Bear` would be missing one of the required methods. In other words, there is no implementation of the `Bear` class that inherits from `Herbivore` and `Omnivore` that the compiler would accept.

The compiler would also throw an exception if you define an `interface` or `abstract` class that inherits from two conflicting interfaces, as shown here:

```
public interface Herbivore {
  public int eatPlants();
}

public interface Omnivore {
  public void eatPlants();
}

public interface Supervore extends Herbivore, Omnivore {} // DOES NOT COMPILE

public abstract class AbstractBear implements Herbivore, Omnivore {}
                                                  // DOES NOT COMPILE
```

Even without implementation details, the compiler detects the problem with the `abstract` definition and prevents compilation.

This concludes our discussion of `abstract` interface methods and multiple inheritance. We'll return to this discussion shortly after we introduce default interface methods. You'll see that things work a bit differently with default interface methods.

Interface Variables

Let's expand our discussion of interfaces to include interface variables, which can be defined within an interface. Like interface methods, interface variables are assumed to be `public`. Unlike interface methods, though, interface variables are also assumed to be `static` and `final`.

Here are two interface variables rules:

1. Interface variables are assumed to be `public`, `static`, and `final`. Therefore, marking a variable as `private` or `protected` will trigger a compiler error, as will marking any variable as `abstract`.
2. The value of an interface variable must be set when it is declared since it is marked as `final`.

In this manner, interface variables are essentially constant variables defined on the interface level. Because they are assumed to be `static`, they are accessible even without

an instance of the interface. Like our earlier `CanFly` example, the following two interface definitions are equivalent, because the compiler will automatically convert them both to the second example:

```
public interface CanSwim {
  int MAXIMUM_DEPTH = 100;
  final static boolean UNDERWATER = true;
  public static final String TYPE = "Submersible";
}

public interface CanSwim {
  public static final int MAXIMUM_DEPTH = 100;
  public static final boolean UNDERWATER = true;
  public static final String TYPE = "Submersible";
}
```

As we see in this example, the compile will automatically insert `public static final` to any constant interface variables it finds missing those modifiers. Also note that it is a common coding practice to use uppercase letters to denote constant values within a class.

Based on these rules, it should come as no surprise that the following entries will not compile:

```
public interface CanDig {
  private int MAXIMUM_DEPTH = 100;  // DOES NOT COMPILE
  protected abstract boolean UNDERWATER = false;  // DOES NOT COMPILE
  public static String TYPE;  // DOES NOT COMPILE
}
```

The first example, `MAXIMUM_DEPTH`, doesn't compile because the `private` modifier is used, and all interface variables are assumed to be `public`. The second line, `UNDERWATER`, doesn't compile for two reasons. It is marked as `protected`, which conflicts with the assumed modifier `public`, and it is marked as `abstract`, which conflicts with the assumed modifier `final`. Finally, the last example, `TYPE`, doesn't compile because it is missing a value. Unlike the other examples, the modifiers are correct, but as you may remember from Chapter 4, you must provide a value to a `static final` member of the class when it is defined.

Default Interface Methods

With the release of Java 8, the authors of Java have introduced a new type of method to an interface, referred to as a default method. A *default method* is a method defined within an interface with the `default` keyword in which a method body is provided. Contrast default methods with "regular" methods in an interface, which are assumed to be abstract and may not have a method body.

A default method within an interface defines an abstract method with a default implementation. In this manner, classes have the option to override the default method if they need to, but they are not required to do so. If the class doesn't override the method, the default implementation will be used. In this manner, the method definition is concrete, not abstract.

The purpose of adding default methods to the Java language was in part to help with code development and backward compatibility. Imagine you have an interface that is shared among dozens or even hundreds of users that you would like to add a new method to. If you just update the interface with the new method, the implementation would break among all of your subscribers, who would then be forced to update their code. In practice, this might even discourage you from making the change altogether. By providing a default implementation of the method, though, the interface becomes backward compatible with the existing codebase, while still providing those individuals who do want to use the new method with the option to override it.

The following is an example of a default method defined in an interface:

```
public interface IsWarmBlooded {
  boolean hasScales();
  public default double getTemperature() {
    return 10.0;
  }
}
```

This example defines two interface methods, one is a normal abstract method and the other a default method. Note that both methods are assumed to be `public`, as all methods of an interface are all `public`. The first method is terminated with a semicolon and doesn't provide a body, whereas the second default method provides a body. Any class that implements `IsWarmBlooded` may rely on the default implementation of `getTemperature()` or override the method and create its own version.

Note that the default access modifier as defined in Chapter 4 is completely different from the default method defined in this chapter. We defined a default access modifier in Chapter 4 as lack of an access modifier, which indicated a class may access a class, method, or value within another class if both classes are within the same package. In this chapter, we are specifically talking about the keyword `default` as applied to a method within an interface. Because all methods within an interface are assumed to be `public`, the access modifier for a default method is therefore `public`.

The following are the default interface method rules you need to be familiar with:

1. A default method may only be declared within an interface and not within a class or abstract class.
2. A default method must be marked with the `default` keyword. If a method is marked as `default`, it must provide a method body.
3. A default method is not assumed to be `static`, `final`, or `abstract`, as it may be used or overridden by a class that implements the interface.

4. Like all methods in an interface, a default method is assumed to be `public` and will not compile if marked as `private` or `protected`.

The first rule should give you some comfort in that you'll only see default methods in interfaces. If you see them in a class on the exam, assume the code will not compile. The second rule just denotes syntax, as default methods must use the `default` keyword. For example, the following code snippets will not compile:

```
public interface Carnivore {
  public default void eatMeat();  // DOES NOT COMPILE
  public int getRequiredFoodAmount() {  // DOES NOT COMPILE
    return 13;
  }
}
```

In this example, the first method, `eatMeat()`, doesn't compile because it is marked as `default` but doesn't provide a method body. The second method, `getRequiredFoodAmount()`, also doesn't compile because it provides a method body but is not marked with the `default` keyword.

Unlike interface variables, which are assumed `static` class members, default methods cannot be marked as `static` and require an instance of the class implementing the interface to be invoked. They can also not be marked as `final` or `abstract`, because they are allowed to be overridden in subclasses but are not required to be overridden.

When an interface `extends` another interface that contains a default method, it may choose to ignore the default method, in which case the default implementation for the method will be used. Alternatively, the interface may override the definition of the default method using the standard rules for method overriding, such as not limiting the accessibility of the method and using covariant returns. Finally, the interface may redeclare the method as abstract, requiring classes that implement the new interface to explicitly provide a method body. Analogous options apply for an abstract class that `implements` an interface.

For example, the following class overrides one default interface method and redeclares a second interface method as `abstract`:

```
public interface HasFins {
  public default int getNumberOfFins() {
    return 4;
  }
  public default double getLongestFinLength() {
    return 20.0;
  }
  public default boolean doFinsHaveScales() {
    return true;
  }
}
```

```
public interface SharkFamily extends HasFins {
  public default int getNumberOfFins() {
    return 8;
  }
  public double getLongestFinLength();
  public boolean doFinsHaveScales() {  // DOES NOT COMPILE
    return false;
  }
}
```

In this example, the first interface, `HasFins`, defines three default methods: `getNumberOfFins()`, `getLongestFinLength()`, and `doFinsHaveScales()`. The second interface, `SharkFamily`, extends `HasFins` and overrides the default method `getNumberOfFins()` with a new method that returns a different value. Next, the `SharkFamily` interface replaces the default method `getLongestFinLength()` with a new abstract method, forcing any class that implements the `SharkFamily` interface to provide an implementation of the method. Finally, the `SharkFamily` interface overrides the `doFinsHaveScales()` method but doesn't mark the method as `default`. Since interfaces may only contain methods with a body that are marked as `default`, the code will not compile.

Because default methods are new to Java 8, there will probably be a few questions on the exam about them, although they likely will not be any more difficult than the previous example.

Default Methods and Multiple Inheritance

You may have realized that by allowing default methods in interfaces, coupled with the fact a class may implement multiple interfaces, Java has essentially opened the door to multiple inheritance problems. For example, what value would the following code output?

```
public interface Walk {
  public default int getSpeed() {
    return 5;
  }
}

public interface Run {
  public default int getSpeed() {
    return 10;
  }
}

public class Cat implements Walk, Run {  // DOES NOT COMPILE
  public static void main(String[] args) {
    System.out.println(new Cat().getSpeed());
  }
}
```

In this example, `Cat` inherits the two default methods for `getSpeed()`, so which does it use? Since `Walk` and `Run` are considered siblings in terms of how they are used in the `Cat` class, it is not clear whether the code should output `5` or `10`. The answer is that the code outputs neither value—it fails to compile.

If a class implements two interfaces that have default methods with the same name and signature, the compiler will throw an error. There is an exception to this rule, though: if the subclass overrides the duplicate default methods, the code will compile without issue—the ambiguity about which version of the method to call has been removed. For example, the following modified implementation of `Cat` will compile and output `1`:

```
public class Cat implements Walk, Run {
  public int getSpeed() {
    return 1;
  }

  public static void main(String[] args) {
    System.out.println(new Cat().getSpeed());
  }
}
```

You can see that having a class that implements or inherits two duplicate default methods forces the class to implement a new version of the method, or the code will not compile. This rule holds true even for abstract classes that implement multiple interfaces, because the default method could be called in a concrete method within the abstract class.

Static Interface Methods

Java 8 also now includes support for static methods within interfaces. These methods are defined explicitly with the `static` keyword and function nearly identically to static methods defined in classes, as discussed in Chapter 4. In fact, there is really only one distinction between a static method in a class and an interface. A static method defined in an interface is not inherited in any classes that implement the interface.

Here are the static interface method rules you need to be familiar with:

1. Like all methods in an interface, a static method is assumed to be `public` and will not compile if marked as `private` or `protected`.
2. To reference the static method, a reference to the name of the interface must be used.

The following is an example of a static method defined in an interface:

```
public interface Hop {
  static int getJumpHeight() {
    return 8;
  }
}
```

The method `getJumpHeight()` works just like a static method as defined in a class. In other words, it can be accessed without an instance of the class using the `Hop.getJumpHeight()` syntax. Also, note that the compiler will automatically insert the access modifier `public` since all methods in interfaces are assumed to be `public`.

The following is an example of a class `Bunny` that implements `Hop`:

```
public class Bunny implements Hop {
  public void printDetails() {
    System.out.println(getJumpHeight()); // DOES NOT COMPILE
  }
}
```

As you can see, without an explicit reference to the name of the interface the code will not compile, even though `Bunny` implements `Hop`. In this manner, the static interface methods are not inherited by a class implementing the interface. The following modified version of the code resolves the issue with a reference to the interface name `Hop`:

```
public class Bunny implements Hop {
  public void printDetails() {
    System.out.println(Hop.getJumpHeight());
  }
}
```

It follows, then, that a class that implements two interfaces containing static methods with the same signature will still compile at runtime, because the static methods are not inherited by the subclass and must be accessed with a reference to the interface name. Contrast this with the behavior you saw for default interface methods in the previous section: the code would compile if the subclass overrode the default methods and would fail to compile otherwise. You can see that static interface methods have none of the same multiple inheritance issues and rules as default interface methods do.

Understanding Polymorphism

Java supports *polymorphism*, the property of an object to take on many different forms. To put this more precisely, a Java object may be accessed using a reference with the same type as the object, a reference that is a superclass of the object, or a reference that defines an interface the object `implements`, either directly or through a superclass. Furthermore, a cast is not required if the object is being reassigned to a super type or interface of the object.

Let's illustrate this polymorphism property with the following example:

```
public class Primate {
  public boolean hasHair() {
    return true;
```

```
  }
}

public interface HasTail {
  public boolean isTailStriped();
}

public class Lemur extends Primate implements HasTail {
  public boolean isTailStriped() {
    return false;
  }
  public int age = 10;
  public static void main(String[] args) {
    Lemur lemur = new Lemur();
    System.out.println(lemur.age);

    HasTail hasTail = lemur;
    System.out.println(hasTail.isTailStriped());

    Primate primate = lemur;
    System.out.println(primate.hasHair());
    }
}
```

This code compiles and executes without issue and yields the following output:

```
10
false
true
```

The most important thing to note about this example is that only one object, `Lemur`, is created and referenced. The ability of an instance of `Lemur` to be passed as an instance of an interface it implements, `HasTail`, as well as an instance of one of its superclasses, `Primate`, is the nature of polymorphism.

Once the object has been assigned a new reference type, only the methods and variables available to that reference type are callable on the object without an explicit cast. For example, the following snippets of code will not compile:

```
HasTail hasTail = lemur;
System.out.println(hasTail.age);  // DOES NOT COMPILE
```

```
Primate primate = lemur;
System.out.println(primate.isTailStriped());  // DOES NOT COMPILE
```

In this example, the reference `hasTail` has direct access only to methods defined with the `HasTail` interface; therefore, it doesn't know the variable `age` is part of the object. Likewise, the reference `primate` has access only to methods defined in the `Primate` class, and it doesn't have direct access to the `isTailStriped()` method.

Object vs. Reference

In Java, all objects are accessed by reference, so as a developer you never have direct access to the object itself. Conceptually, though, you should consider the object as the entity that exists in memory, allocated by the Java runtime environment. Regardless of the type of the reference you have for the object in memory, the object itself doesn't change. For example, since all objects inherit `java.lang.Object`, they can all be reassigned to `java.lang.Object`, as shown in the following example:

```
Lemur lemur = new Lemur();

Object lemurAsObject = lemur;
```

Even though the `Lemur` object has been assigned a reference with a different type, the object itself has not changed and still exists as a `Lemur` object in memory. What has changed, then, is our ability to access methods within the `Lemur` class with the `lemurAsObject` reference. Without an explicit cast back to `Lemur`, as you'll see in the next section, we no longer have access to the `Lemur` properties of the object.

We can summarize this principle with the following two rules:

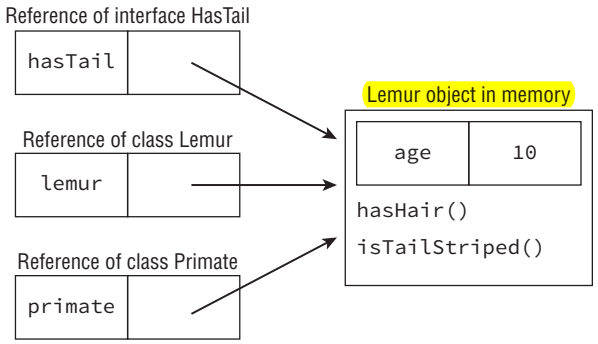
1. The type of the object determines which properties exist within the object in memory.
2. The type of the reference to the object determines which methods and variables are accessible to the Java program.

It therefore follows that successfully changing a reference of an object to a new reference type may give you access to new properties of the object, but those properties existed before the reference change occurred.

Let's illustrate this property using the previous example in Figure 5.6. As you can see in the figure, the same object exists in memory regardless of which reference is pointing to it. Depending on the type of the reference, we may only have access to certain methods. For example, the `hasTail` reference has access to the method `isTailStriped()` but doesn't have access to the variable `age` defined in the `Lemur` class. As you'll learn in the next section, it is

possible to reclaim access to the variable age by explicitly casting the hasTail reference to a reference of type Lemur.

FIGURE 5.6 Object vs. reference



Casting Objects

In the previous example, we created a single instance of a Lemur object and accessed it via superclass and interface references. Once we changed the reference type, though, we lost access to more specific methods defined in the subclass that still exist within the object. We can reclaim those references by casting the object back to the specific subclass it came from:

```
Primate primate = lemur;
Lemur lemur2 = primate; // DOES NOT COMPILE

Lemur lemur3 = (Lemur)primate;
System.out.println(lemur3.age);
```

In this example, we first try to convert the primate reference back to a lemur reference, lemur2, without an explicit cast. The result is that the code will not compile. In the second example, though, we explicitly cast the object to a subclass of the object Primate and we gain access to all the methods available to the Lemur class.

Here are some basic rules to keep in mind when casting variables:

1. Casting an object from a subclass to a superclass doesn't require an explicit cast.
2. Casting an object from a superclass to a subclass requires an explicit cast.
3. The compiler will not allow casts to unrelated types.
4. Even when the code compiles without issue, an exception may be thrown at runtime if the object being cast is not actually an instance of that class.

The third rule is important; the exam may try to trick you with a cast that the compiler doesn't allow. For example, we were able to cast a `Primate` reference to a `Lemur` reference, because `Lemur` is a subclass of `Primate` and therefore related.

Consider this example:

```
public class Bird {}

public class Fish {
  public static void main(String[] args) {
    Fish fish = new Fish();
    Bird bird = (Bird)fish;  // DOES NOT COMPILE
  }
}
```

In this example, the classes `Fish` and `Bird` are not related through any class hierarchy that the compiler is aware of; therefore, the code will not compile.

Casting is not without its limitations. Even though two classes share a related hierarchy, that doesn't mean an instance of one can automatically be cast to another. Here's an example:

```
public class Rodent {
}

public class Capybara extends Rodent {
  public static void main(String[] args) {
    Rodent rodent = new Rodent();
    Capybara capybara = (Capybara)rodent; // Throws ClassCastException at runtime
  }
}
```

This code creates an instance of `Rodent` and then tries to cast it to a subclass of `Rodent`, `Capybara`. Although this code will compile without issue, it will throw a `ClassCastException` at runtime since the object being referenced is not an instance of the `Capybara` class. The thing to keep in mind in this example is the object that was created is not related to the `Capybara` class in any way.

Although this topic is out of scope for the OCA exam, keep in mind that the `instanceof` operator can be used to check whether an object belongs to a particular class and to prevent `ClassCastException`s at runtime. Unlike the previous example, the following code snippet doesn't throw an exception at runtime and performs the cast only if the `instanceof` operator returns `true`.

```
if(rodent instanceof Capybara) {
  Capybara capybara = (Capybara)rodent;
}
```

When reviewing a question on the exam that involves casting and polymorphism, be sure to remember what the instance of the object actually is. Then, focus on whether the compiler will allow the object to be referenced with or without explicit casts.

Virtual Methods

The most important feature of polymorphism—and one of the primary reasons we have class structure at all—is to support virtual methods. A *virtual method* is a method in which the specific implementation is not determined until runtime. In fact, all non-final, non-static, and non-private Java methods are considered virtual methods, since any of them can be overridden at runtime. What makes a virtual method special in Java is that if you call a method on an object that overrides a method, you get the overridden method, even if the call to the method is on a parent reference or within the parent class.

We'll illustrate this principle with the following example:

```
public class Bird {
  public String getName() {
    return "Unknown";
  }
  public void displayInformation() {
    System.out.println("The bird name is: "+getName());
  }
}

public class Peacock extends Bird {
  public String getName() {
    return "Peacock";
  }
  public static void main(String[] args) {
    Bird bird = new Peacock();
    bird.displayInformation();
  }
}
```

This code compiles and executes without issue and outputs the following:

```
The bird name is: Peacock
```

As you saw in similar examples in the section "Overriding a Method," the method `getName()` is overridden in the child class `Peacock`. More importantly, though, the value of the `getName()` method at runtime in the `displayInformation()` method is replaced with the value of the implementation in the subclass `Peacock`.

In other words, even though the parent class `Bird` defines its own version of `getName()` and doesn't know anything about the `Peacock` class during compile-time, at runtime the instance uses the overridden version of the method, as defined on the instance of the object. We emphasize this point by using a reference to the `Bird` class in the `main()` method, although the result would have been the same if a reference to `Peacock` was used.

You now know the true purpose of overriding a method and how it relates to polymorphism. The nature of the polymorphism is that an object can take on many different forms. By combining your understanding of polymorphism with method overriding, you see that objects may be interpreted in vastly different ways at runtime, especially in methods defined in the superclasses of the objects.

Polymorphic Parameters

One of the most useful applications of polymorphism is the ability to pass instances of a subclass or interface to a method. For example, you can define a method that takes an instance of an interface as a parameter. In this manner, any class that implements the interface can be passed to the method. Since you're casting from a subtype to a supertype, an explicit cast is not required. This property is referred to as *polymorphic parameters* of a method, and we demonstrate it in the following example:

```
public class Reptile {
  public String getName() {
    return "Reptile";
  }
}

public class Alligator extends Reptile {
  public String getName() {
    return "Alligator";
  }
}

public class Crocodile extends Reptile {
  public String getName() {
    return "Crocodile";
  }
}

public class ZooWorker {
  public static void feed(Reptile reptile) {
    System.out.println("Feeding reptile "+reptile.getName());
  }
```

```
  public static void main(String[] args) {
    feed(new Alligator());
    feed(new Crocodile());
    feed(new Reptile());
  }
}
```

This code compiles and executes without issue, yielding the following output:

```
Feeding: Alligator
Feeding: Crocodile
Feeding: Reptile
```

Let's focus on the `feed(Reptile reptile)` method in this example. As you can see, that method was able to handle instances of `Alligator` and `Crocodile` without issue, because both are subclasses of the `Reptile` class. It was also able to accept a matching type `Reptile` class. If we had tried to pass an unrelated class, such as the previously defined `Rodent` or `Capybara` classes, or a superclass such as `java.lang.Object`, to the `feed()` method, the code would not have compiled.

Polymorphic Parameters and Code Reusability

If you're defining a method that will be accessible outside the current class, either to subclasses of the current class or publicly to objects outside the current class, it is considered good coding practice to use the superclass or interface type of input parameters whenever possible.

As you may remember from Chapter 3, "Core Java APIs," the type `java.util.List` is an interface, not a class. Although there are many classes that implement `java.util.List`, such as `java.util.ArrayList` and `java.util.Vector`, when you're passing an existing `List` you're not usually interested in the particular subclass of the `List`. In this manner, a method that passes a `List` should use the interface type `java.util.List` as the polymorphic parameter type, rather than a specific class that implements `List`, as the code will be more reusable for other types of lists.

For example, it is common to see code such as the following that uses the interface reference type over the class type for greater reusability:

```
java.util.List list = new java.util.ArrayList();
```

Polymorphism and Method Overriding

Let's conclude this chapter by returning to the last three rules for method overriding to demonstrate how polymorphism requires them to be included as part of the Java specification. You'll see that without such rules in place, it is easy to construct an example with polymorphism in Java.

The first rule is that an overridden method must be at least as accessible as the method it is overriding. Let's assume this rule is not necessary and consider the following example:

```
public class Animal {
  public String getName() {
    return "Animal";
  }
}

public class Gorilla extends Animal {
  protected String getName() {  // DOES NOT COMPILE
    return "Gorilla";
  }
}

public class ZooKeeper {
  public static void main(String[] args) {
    Animal animal = new Gorilla();
    System.out.println(animal.getName());
  }
}
```

For the purpose of this discussion, we'll ignore the fact that the implementation of `getName()` in the `Gorilla` class doesn't compile because it is less accessible than the version it is overriding in the `Animal` class.

As you can see, this example creates an ambiguity problem in the `ZooKeeper` class. The reference `animal.getName()` is allowed because the method is `public` in the `Animal` class, but due to polymorphism, the `Gorilla` object itself has been overridden with a less accessible version, not available to the `ZooKeeper` class. This creates a contradiction in that the compiler should not allow access to this method, but because it is being referenced as an instance of `Animal`, it is allowed. Therefore, Java eliminates this contradiction, thus disallowing a method from being overridden by a less accessible version of the method.

Likewise, a subclass cannot declare an overridden method with a new or broader exception than in the superclass, since the method may be accessed using a reference to the superclass. For example, if an instance of the subclass is passed to a method using a superclass reference, then the enclosing method would not know about any new checked exceptions that exist on methods for this object, potentially leading to compiled code with

"unchecked" checked exceptions. Therefore, the Java compiler disallows overriding methods with new or broader exceptions.

Finally, overridden methods must use covariant return types for the same kinds of reasons as just discussed. If an object is cast to a superclass reference and the overridden method is called, the return type must be compatible with the return type of the parent method. If the return type in the child is too broad, it will result an inherent cast exception when accessed through the superclass reference.

For example, if the return type of a method is `Double` in the parent class and is overridden in a subclass with a method that returns `Number`, a superclass of `Double`, then the subclass method would be allowed to return any valid `Number`, including `Integer`, another subclass of `Number`. If we are using the object with a reference to the superclass, that means an `Integer` could be returned when a `Double` was expected. Since `Integer` is not a subclass of `Double`, this would lead to an implicit cast exception as soon as the value was referenced. Java solves this problem by only allowing covariant return types for overridden methods.

Summary

This chapter took the basic class structure we presented in Chapter 4 and expanded it by introducing the notion of inheritance. Java classes follow a multilevel single-inheritance pattern in which every class has exactly one direct parent class, with all classes eventually inheriting from `java.lang.Object`. Java interfaces simulate a limited form of multiple inheritance, since Java classes may implement multiple interfaces.

Inheriting a class gives you access to all of the `public` and `protected` methods of the class, but special rules for constructors and overriding methods must be followed or the code will not compile. For example, if the parent class doesn't include a no-argument constructor, an explicit call to a parent constructor must be provided in the child's constructors. Pay close attention on the exam to any class that defines a constructor with arguments and doesn't define a no-argument constructor.

We reviewed overloaded, overridden, and hidden methods and showed how they differ, especially in terms of polymorphism. We also introduced the notion of hiding variables, although we strongly discourage this in practice as it often leads to confusing, difficult-to-maintain code.

We introduced abstract classes and interfaces and showed how you can use them to define a platform for other developers to interact with. By definition, an abstract type cannot be instantiated directly and requires a concrete subclass for the code to be used. Since `default` and `static` interface methods are new to Java 8, expect to see at least one question on them on the exam.

Finally, this chapter introduced the concept of polymorphism, central to the Java language, and showed how objects can be accessed in a variety of forms. Make sure you understand when casts are needed for accessing objects, and be able to spot the difference between compile-time and runtime cast problems.

Exam Essentials

Be able to write code that extends other classes. A Java class that extends another class inherits all of its `public` and `protected` methods and variables. The first line of every constructor is a call to another constructor within the class using `this()` or a call to a constructor of the parent class using the `super()` call. If the parent class doesn't contain a no-argument constructor, an explicit call to the parent constructor must be provided. Parent methods and objects can be accessed explicitly using the `super` keyword. Finally, all classes in Java extend `java.lang.Object` either directly or from a superclass.

Understand the rules for method overriding. The Java compiler allows methods to be overridden in subclasses if certain rules are followed: a method must have the same signature, be at least as accessible as the parent method, must not declare any new or broader exceptions, and must use covariant return types.

Understand the rules for hiding methods and variables. When a static method is recreated in a subclass, it is referred to as method hiding. Likewise, variable hiding is when a variable name is reused in a subclass. In both situations, the original method or variable still exists and is used in methods that reference the object in the parent class. For method hiding, the use of `static` in the method declaration must be the same between the parent and child class. Finally, variable and method hiding should generally be avoided since it leads to confusing and difficult-to-follow code.

Recognize the difference between method overriding and method overloading. Both method overloading and overriding involve creating a new method with the same name as an existing method. When the method signature is the same, it is referred to as method overriding and must follow a specific set of override rules to compile. When the method signature is different, with the method taking different inputs, it is referred to as method overloading and none of the override rules are required.

Be able to write code that creates and extends abstract classes. In Java, classes and methods can be declared as `abstract`. Abstract classes cannot be instantiated and require a concrete subclass to be accessed. Abstract classes can include any number, including zero, of abstract and nonabstract methods. Abstract methods follow all the method override rules and may only be defined within abstract classes. The first concrete subclass of an abstract class must implement all the inherited methods. Abstract classes and methods may not be marked as `final` or `private`.

Be able to write code that creates, extends, and implements interfaces. Interfaces are similar to a specialized abstract class in which only abstract methods and constant `static final` variables are allowed. New to Java 8, an interface can also define default and static methods with method bodies. All members of an interface are assumed to be public. Methods are assumed to be abstract if not explicitly marked as `default` or `static`. An interface that extends another interface inherits all its abstract methods. An interface cannot extend a class, nor can a class extend an interface. Finally, classes may implement any number of interfaces.

Be able to write code that uses default and static interface methods. A default method allows a developer to add a new method to an interface used in existing implementations, without forcing other developers using the interface to recompile their code. A developer using the interface may override the default method or use the provided one. A static method in an interface follows the same rules for a static method in a class.

Understand polymorphism. An object in Java may take on a variety of forms, in part depending on the reference used to access the object. Methods that are overridden will be replaced everywhere they are used, whereas methods and variables that are hidden will only be replaced in the classes and subclasses that they are defined. It is common to rely on polymorphic parameters—the ability of methods to be automatically passed as a superclass or interface reference—when creating method definitions.

Recognize valid reference casting. An instance can be automatically cast to a superclass or interface reference without an explicit cast. Alternatively, an explicit cast is required if the reference is being narrowed to a subclass of the object. The Java compiler doesn't permit casting to unrelated types. You should be able to discern between compiler-time casting errors and those that will not occur until runtime and that throw a `CastClassException`.

Review Questions

1. What modifiers are implicitly applied to all interface methods? (Choose all that apply)
   A. `protected`
   B. `public`
   C. `static`
   D. `void`
   E. `abstract`
   F. `default`

2. What is the output of the following code?

```
1: class Mammal {
2:   public Mammal(int age) {
3:     System.out.print("Mammal");
4:   }
5: }
6: public class Platypus extends Mammal {
7:   public Platypus() {
8:     System.out.print("Platypus");
9:   }
10:  public static void main(String[] args) {
11:    new Mammal(5);
12:  }
13: }
```

   A. `Platypus`
   B. `Mammal`
   C. `PlatypusMammal`
   D. `MammalPlatypus`
   E. The code will not compile because of line 8.
   F. The code will not compile because of line 11.

3. Which of the following statements can be inserted in the blank line so that the code will compile successfully? (Choose all that apply)

```
public interface CanHop {}
public class Frog implements CanHop {
   public static void main(String[] args) {
     _____ frog = new TurtleFrog();
   }
}
```

```
public class BrazilianHornedFrog extends Frog {}
public class TurtleFrog extends Frog {}
```

A. `Frog`

B. `TurtleFrog`

C. `BrazilianHornedFrog`

D. `CanHop`

E. `Object`

F. `Long`

4. Which statement(s) are correct about the following code? (Choose all that apply)

```
public class Rodent {
  protected static Integer chew() throws Exception {
    System.out.println("Rodent is chewing");
    return 1;
  }
}
public class Beaver extends Rodent {
  public Number chew() throws RuntimeException {
    System.out.println("Beaver is chewing on wood");
    return 2;
  }
}
```

A. It will compile without issue.

B. It fails to compile because the type of the exception the method throws is a subclass of the type of exception the parent method throws.

C. It fails to compile because the return types are not covariant.

D. It fails to compile because the method is `protected` in the parent class and `public` in the subclass.

E. It fails to compile because of a `static` modifier mismatch between the two methods.

5. Which of the following may only be hidden and not overridden? (Choose all that apply)

A. `private` instance methods

B. `protected` instance methods

C. `public` instance methods

D. `static` methods

E. `public` variables

F. `private` variables

6. Choose the correct statement about the following code:

```
1: interface HasExoskeleton {
2:   abstract int getNumberOfSections();
3: }
4: abstract class Insect implements HasExoskeleton {
5:   abstract int getNumberOfLegs();
6: }
7: public class Beetle extends Insect {
8:   int getNumberOfLegs() { return 6; }
9: }
```

   A. It compiles and runs without issue.
   B. The code will not compile because of line 2.
   C. The code will not compile because of line 4.
   D. The code will not compile because of line 7.
   E. It compiles but throws an exception at runtime.

7. Which of the following statements about polymorphism are true? (Choose all that apply)
   A. A reference to an object may be cast to a subclass of the object without an explicit cast.
   B. If a method takes a superclass of three objects, then any of those classes may be passed as a parameter to the method.
   C. A method that takes a parameter with type `java.lang.Object` will take any reference.
   D. All cast exceptions can be detected at compile-time.
   E. By defining a public instance method in the superclass, you guarantee that the specific method will be called in the parent class at runtime.

8. Choose the correct statement about the following code:

```
1: public interface Herbivore {
2:   int amount = 10;
3:   public static void eatGrass();
4:   public int chew() {
5:     return 13;
6:   }
7: }
```

   A. It compiles and runs without issue.
   B. The code will not compile because of line 2.
   C. The code will not compile because of line 3.
   D. The code will not compile because of line 4.
   E. The code will not compile because of lines 2 and 3.
   F. The code will not compile because of lines 3 and 4.

9. Choose the correct statement about the following code:

```
1: public interface CanFly {
2:   void fly();
3: }
4: interface HasWings {
5:   public abstract Object getWindSpan();
6: }
7: abstract class Falcon implements CanFly, HasWings {
8: }
```

A. It compiles without issue.

B. The code will not compile because of line 2.

C. The code will not compile because of line 4.

D. The code will not compile because of line 5.

E. The code will not compile because of lines 2 and 5.

F. The code will not compile because the class `Falcon` doesn't implement the interface methods.

10. Which statements are true for both abstract classes and interfaces? (Choose all that apply)

A. All methods within them are assumed to be abstract.

B. Both can contain `public static final` variables.

C. Both can be extended using the `extend` keyword.

D. Both can contain default methods.

E. Both can contain static methods.

F. Neither can be instantiated directly.

G. Both inherit `java.lang.Object`.

11. What modifiers are assumed for all interface variables? (Choose all that apply)

A. `public`

B. `protected`

C. `private`

D. `static`

E. `final`

F. `abstract`

12. What is the output of the following code?

```
1: interface Nocturnal {
2:   default boolean isBlind() { return true; }
3: }
4: public class Owl implements Nocturnal {
```

```
5:   public boolean isBlind() { return false; }
6:     public static void main(String[] args) {
7:     Nocturnal nocturnal = (Nocturnal)new Owl();
8:     System.out.println(nocturnal.isBlind());
9:     }
10: }
```

A. `true`

B. `false`

C. The code will not compile because of line 2.

D. The code will not compile because of line 5.

E. The code will not compile because of line 7.

F. The code will not compile because of line 8.

13. What is the output of the following code?

```
1: class Arthropod
2:   public void printName(double input) { System.out
  .print("Arthropod"); }
3: }
4: public class Spider extends Arthropod {
5:   public void printName(int input) { System.out.print("Spider"); }
6:   public static void main(String[] args) {
7:     Spider spider = new Spider();
8:     spider.printName(4);
9:     spider.printName(9.0);
10:   }
11: }
```

A. `SpiderArthropod`

B. `ArthropodSpider`

C. `SpiderSpider`

D. `ArthropodArthropod`

E. The code will not compile because of line 5.

F. The code will not compile because of line 9.

14. Which statements are true about the following code? (Choose all that apply)

```
1: interface HasVocalCords {
2:   public abstract void makeSound();
3: }
4: public interface CanBark extends HasVocalCords {
5:   public void bark();
6: }
```

A. The `CanBark` interface doesn't compile.

B. A class that implements `HasVocalCords` must override the `makeSound()` method.

C. A class that implements `CanBark` inherits both the `makeSound()` and `bark()` methods.

D. A class that implements `CanBark` only inherits the `bark()` method.

E. An interface cannot extend another interface.

15. Which of the following is true about a concrete subclass? (Choose all that apply)

A. A concrete subclass can be declared as `abstract`.

B. A concrete subclass must implement all inherited abstract methods.

C. A concrete subclass must implement all methods defined in an inherited interface.

D. A concrete subclass cannot be marked as `final`.

E. Abstract methods cannot be overridden by a concrete subclass.

16. What is the output of the following code?

```
1: abstract class Reptile {
2:   public final void layEggs() { System.out.println("Reptile laying eggs");
   }
3:     public static void main(String[] args) {
4:     Reptile reptile = new Lizard();
5:     reptile.layEggs();
6:     }
7: }
8: public class Lizard extends Reptile {
9:   public void layEggs() { System.out.println("Lizard laying eggs"); }
10: }
```

A. `Reptile laying eggs`

B. `Lizard laying eggs`

C. The code will not compile because of line 4.

D. The code will not compile because of line 5.

E. The code will not compile because of line 9.

17. What is the output of the following code?

```
1: public abstract class Whale {
2:   public abstract void dive() {};
3:   public static void main(String[] args) {
4:     Whale whale = new Orca();
5:     whale.dive();
6:   }
7: }
```

```
8: class Orca extends Whale {
9:   public void dive(int depth) { System.out.println("Orca diving"); }
10: }
```

A. `Orca diving`

B. The code will not compile because of line 2.

C. The code will not compile because of line 8.

D. The code will not compile because of line 9.

E. The output cannot be determined from the code provided.

18. What is the output of the following code? (Choose all that apply)

```
1: interface Aquatic {
2:   public default int getNumberOfGills(int input) { return 2; }
3: }
4: public class ClownFish implements Aquatic {
5:   public String getNumberOfGills() { return "4"; }
6:   public String getNumberOfGills(int input) { return "6"; }
7:   public static void main(String[] args) {
8:     System.out.println(new ClownFish().getNumberOfGills(-1));
9:   }
10: }
```

A. 2

B. 4

C. 6

D. The code will not compile because of line 5.

E. The code will not compile because of line 6.

F. The code will not compile because of line 8.

19. Which of the following statements can be inserted in the blank so that the code will compile successfully? (Choose all that apply)

```
public class Snake {}
public class Cobra extends Snake {}
public class GardenSnake {}
public class SnakeHandler {
   private Snake snake;
   public void setSnake(Snake snake) { this.snake = snake; }
   public static void main(String[] args) {
     new SnakeHandler().setSnake(__________);
   }
}
```

A. `new Cobra()`

B. `new GardenSnake()`

C. `new Snake()`

D. `new Object()`

E. `new String("Snake")`

F. `null`

20. What is the result of the following code?

```
1: public abstract class Bird {
2:   private void fly() { System.out.println("Bird is flying"); }
3:   public static void main(String[] args) {
4:     Bird bird = new Pelican();
5:     bird.fly();
6:   }
7: }
8: class Pelican extends Bird {
9:   protected void fly() { System.out.println("Pelican is flying"); }
10: }
```

A. `Bird is flying`

B. `Pelican is flying`

C. The code will not compile because of line 4.

D. The code will not compile because of line 5.

E. The code will not compile because of line 9.